Assisted Resource Management in the New Zealand Rural Fire Service

Final Report

Andrew Curtis-Black (student), Dr. Andreas Willig (supervisor)

Word count: 13,729 (11,567 body, 2,162 appendices)

Abstract

There exists a need for more efficient tools and processes for use in the emergency services. Of particular interest to the authors of this report and to our industry sponsor, Tait Communications, is the idea of resource management. Resources retained by the emergency services include vehicles, personnel, specialist equipment and tools, and medical and disaster relief supplies. The nature of an emergency mandates a swift response, and thus it is necessary for such resources to be maintained in a state of constant readiness. Existing processes for achieving this are manual and often incur high overheads.

In this report we detail our creation of a resource management application for use in the New Zealand Rural Fire Service. The application was created as a prototype for demonstrating the potential of such a system in order to facilitate further discussion with interested parties. We approached this project with a complete software engineering product lifecycle, from consultation, planning and risk analysis to prototyping, implementation and evaluation.

We produced a functional prototype which was positively received by our industry sponsor, who acted as a proxy customer for the purposes of the project. We thus concluded that our work was successful, and have conclusively demonstrated the utility of a resource management application for use in the emergency services.

Introduction

Purpose

The New Zealand emergency services use a wide variety of assets to carry out their work. These may include medical supplies, vehicles, personnel, fuel, disaster relief supplies, construction materials and tools, food, water, personal safety equipment, and many others. These assets are used to respond to emergencies and as such need to be ready for use at all times. Current processes for ensuring asset readiness are manual and must be repeated often. Different emergency service organisations are responsible for their own assets. During a response to an emergency these assets are deployed and used as needed, rightly with little regard for their jurisdictional affiliations. After an emergency incident has been resolved it is important that these assets be returned to the organisations which are responsible for them. However, assets are commonly misplaced and sometimes lost.

We observed existing processes (both manual and automated) and noted inefficiencies in their design and application. With this project we aim to contribute to the improvement of resource management in the emergency services. We provide a working prototype of a resource management application for use in the New Zealand Rural Fire Service, the purpose of which is to demonstrate the potential of such software. It is our belief that this will encourage further investment in this area of work and lead to greater interest in supporting the emergency services through software engineering. In this report we also provide a description and analysis of the technical processes used to design and implement this application. The essential problems addressed by our work are common to many emergency service groups. While we focus on the New Zealand rural fire service in this project our work could be repurposed for many similar organisations around the world.

Motivation

The emergency services carry out essential work. Supporting their operations with more efficient, modern, and humane processes is a worthwhile endeavour which we believe will make a modest contribution to the greater good of this country. We also note that few of the volunteers and professionals who make up the emergency services are motivated by a desire to perform repetitive administration work, and yet much of their effort is thus channeled.

Objectives

We identified the following items as objectives for our project. Each of these corresponds to an implementable feature to be demonstrated by the prototype we developed.

- Allow users to view the contents of tracked vehicles.
- Present users with a list of incidents (past and present).
- Present users with the locations of tracked vehicles.
- Allow users to search for incidents.
- Automatically log data.

Overview of Chapters

1. The Problem

In this chapter we go into greater detail about the problem this project aims to deal with. We discuss the relationship between this project and a number of others operating in parallel in the Electrical and Computer Engineering department. We describe the research we carried out into the background of the problem, develop a problem statement and project requirements (functional and non-functional), and finally derive a set of formal project objectives.

2. Approaches to Solving the Problem

Here we identify three possible approaches to the problem and select one, offering a discussion of our rationale. We detail our selection of a platform, our approach to risk management, and clearly define the scope of the project.

3. Design

We here present our progression from low to high fidelity prototypes and the observations we made with them.

4. Implementation

In this section we discuss the design of our system's architecture, and the way it interacts with the greater system created in cooperation with the ECE team.

5. Testing

We discuss our approach to testing, delineating between our testing methodology at different stages of the project. We also describe our position on automated GUI testing and continuous integration.

6. Tools and Technologies

In this chapter we describe the tools and technologies we used to develop the application, identifying a range of language features, frameworks and concepts. We also give consideration to the tools we used for administrating the project and describe the ways in which they supported our chosen project management methodology.

7. Evaluation

We critically evaluate our project plan, using hindsight to identify flaws and to suggest possible improvements. We also evaluate the application itself, with reference to its suitability for its intended purpose. Finally, we evaluate our collaboration with the ECE team, identifying both strong and weak points in our relationship.

8. Discussion and Conclusions

Here we assess the extent to which each of the objectives we defined for the project were successfully completed. We likewise assess our product's compliance with the project's functional and non-functional requirements. We identify areas of learning, with reference to specific examples. Finally, we present a number of avenues for future work on the project in the form of recommended features which are accompanied by example user stories for added clarification.

1. The Problem

1.1. Project Background

In the course of normal interactions with the New Zealand emergency services Tait Communications noted a need for resource management software. Many existing processes are manual and paper-based, translating to greater costs and more frequent errors. A four person team of students from the electrical and computer engineering department (ECE) worked on a related project which aims to create a vehicle-area network and attendant services for emergency vehicles such as fire engines [1]. This project makes use of the data gathered by such a network as well as the communication pathways it offers.

1.2. Relationship to ECE Team

There was a group of electrical and computer engineering (ECE) students working as a team on four related projects. These projects mostly centred on gathering data on fire assets (such as vehicle metrics, inventories etc.) Each of the four ECE students had their own projects which formed subsets of responsibilities within the larger scope of the overall project. Thus, we developed our solutions independently but collaborated wherever possible.

To ensure interoperability we agreed on common data formats and methods of communication between this project and the ECE team's project. The primary vehicle for this was through joint authorship of an API for reading and writing data to a common database. One member of the ECE team (Jeffrey Torres) was the contact-person for this part of the project and contributed development time to it.

All parties retained enough independence to tolerate changing requirements in another's project.

1.3. Background Research

Two interviews were held with volunteer firefighters. These yielded a number of insights including an understanding of the workflow of an incident; the identification of a number of perceived problems in the workflow; a list of example assets (such as fire engines, hydraulic cutters, and medical equipment); a preliminary discussion of the command structure and the organisation of a fire fighting work unit (known as a crew); some examples of the domain's vernacular; and a list of the types of common fires. A summary of this information is available in Appendix 3.

From a report produced for Tait and the New Zealand Fire Service a more detailed overview of an incident workflow was obtained, as well as a number of concepts for support software (such as an automatic logbook and a resource management utility). The specific contents of this report remain confidential and thus cannot be reproduced here, or discussed in detail.

As part of our investigations into the problem space of the project we visited FireCom (the centre which coordinates all firefighting efforts in the South Island) and the Rangiora fire depot (which holds a large amount

of firefighting equipment of the type our application will be tracking), interviewing potential users of our application and reviewing previous work done on this problem as we did so.

At FireCom we observed a number of real emergencies in progress and the workflows for handling them. Currently the fire service uses a software application which runs on Windows XP and which offers few of the conveniences of modern applications. With this application they are able to view the locations of vehicles and emergencies on a map which is periodically updated. The application can also display a list of assets which are expected to be present on a vehicle, but this is not updated and serves more as a reference than as a means of assessing asset availability. The application appeared quite inefficient to use, incurring a high interaction cost and offering few affordances to the user. We would expect it to take quite some time for a new user to become proficient with the program. Note that for reasons of confidentiality we are unable to provide screenshots or depictions of the program.

At the Rangiora fire depot we saw the ways in which equipment is stored, both in the depot and on vehicles. We also discussed difficulties currently encountered by staff (eg: inefficient manual logging processes). Summaries of two of the interviews we conducted are included in Appendix 3. See Appendices 1 and 2 for sample checklists used to (manually) ensure that all necessary equipment is loaded onto vehicles which are on standby. These materials helped define the context in which our program will be used, and gave us some insight into potential users' requirements. See Appendix 4 for a series of photos taken at the Rangiora fire depot.

1.4. Problem Statement

Resource[1] management in the emergency services involves regularly auditing the contents of vehicles; allocating work units (known as brigades in the fire service) to incidents; maintaining resources (e.g.: repairing hoses, refilling oxygen tanks); tracking the location of resources; and tracking the use of items. These activities are most often carried out with manual processes which, as they are frequently performed, contribute to resource wastage. The emergency services also lack a centralised source of information on the disposition of their resources and instead rely on a collection of disconnected processes.

A need has been identified for a resource management application which aids individuals managing incidents by automating these procedures and by providing a common point of reference for information relating to resources.

1.5. Aim

This project aimed to produce a mobile application for the iOS platform which demonstrates the potential of the concept outlined above. It did not aim to create a fully functional product, ready for deployment, but rather an application prototype.

[1] Resources may include (but are not limited to) vehicles, tools, medical supplies, food, water, and safety equipment.

1.6. Requirements

The functional requirements listed below are given in order of descending priority. Numbers 1) to 5) are considered the primary goals of the project, with the remaining items forming a basis for potential expansion. All of the non-functional requirements are considered essential to the project, both in terms of defining goals and in limiting scope.

1.6.1. Functional Requirements

1. The application shall allow users at Firecom to view the contents of tracked vehicles.
2. The application shall present incident commanders with a list of incidents. It must be possible to view completed (past) incidents as well as active (current) ones.
3. The application shall automatically log resource data, mirroring/replacing existing manual logging processes which are carried out by staff at fire depots. Examples: Distance travelled by vehicles, vehicles attending a job.
4. The application shall present incident commanders with the locations of tracked assets.
5. Administration staff shall be able to search for tracked incidents with various parameters. Examples: Job number, requirements, equipment usage.
6. The application shall be capable of generating a status report for an active incident. This could be used by an incident manager to quickly appraise themselves of the situation prior to their arrival.
7. It shall be possible to send a range of alerts/messages to tracked vehicles. Example: “Reassigned to incident #111”
8. Incident commanders shall be able to view metrics for incidents, resources, and work units (e.g.: brigades). Examples: Total distance traveled by all vehicles, fuel consumption, total cost of incident.
9. The application shall be capable of suggesting which resources an incident commander should allocate to an incident (based on location, incident requirements, resource readiness, job urgency etc.)

1.6.2. Non-Functional Requirements

- The application shall be suitable for use by an incident commander (IC), as in the New Zealand Rural Fire Service. The IC may use the application to gather information about the incident prior to arriving at the scene, to monitor resources during an incident, to request additional resources, and/or to review an incident after it has finished.
- The project aims to improve the operations of the clients, not to introduce software which hinders or complicates existing processes.
- The application shall be a concept related to rural fire used to facilitate discussion for further feedback.
- The application shall be user friendly
  - This may be evaluated by carrying out user testing
- The application shall run on an iPad and be produced using Swift, Cocoa Touch and/or Objective C
- The application shall be self-contained. It shall not require extensive integration with any existing systems in order to add value or otherwise be judged successful.

- Though the application is intended for use in stressful situations, for the purposes of this project this shall not be a primary concern. Making the application robust to user error while under stress could require a prohibitive commitment of time and effort.
  - User testing should be limited to ensuring that the user interface is easy to understand and use so that it may better demonstrate the capabilities of the system.

1.7. Project Objectives

From the requirements listed above we created a set of project objectives to focus our efforts. These were deliberately written in such a way as to facilitate their translation into Scrum-style user stories (though no direct one-to-one mapping was ever intended).

Table 1. Project Objectives

Objective	Motivation	Classifcation
Allow users to view the contents of tracked vehicles.	Vehicle contents tracking has been identified as an important capability which is not provided by existing systems.	Essential
Present users with a list of incidents (past and present).	Incidents are an existing conceptual component of the workflow used by workers in the emergency services. Including this concept in our application will make it more intuitive, and also provide a convenient navigational tool.	Important
Automatically log data.	Existing manual processes are time consuming and imprecise.	Important
Present users with the locations of tracked vehicles.	This information is critical to users and is provided by existing systems.	Important
Allow users to search for incidents.	The program will track more incidents than can be practically searched manually.	Important
Generate status reports for incidents on demand.	Users may not have time to acquaint themselves with all the details of an incident. Status reports would provide users with a way of quickly appraising a situation, and of exporting information from the application.	Stretch goal
Be capable of sending alerts to tracked vehicles.	Bidirectional communication would empower users to respond to the observations they make with the application.	Stretch goal
Provide graphical and textual summaries of relevant metrics.	Data collected by the application will be most useful to users through real-time visualisations.	Stretch goal
Provide suggestions for resource allocation based on automatic analysis.	This will assist with resource allocation, which can be a highly complex task dependent on many factors.	Stretch goal

2. Approaches to Solving the Problem

2.1. Project Definition

From our background research and discussions with project stakeholders three broad possibilities for the project emerged.

1) A mobile application used to ensure that vehicles have all the equipment they should have before and after jobs. This would not necessarily be used by firefighters, but might be used by staff at depots.

2) A mobile application used by personnel in the field to find equipment in non-ideal circumstances (eg: directly after responding to an emergency).

3) An application (which could be produced for either a mobile or a desktop platform) used by incident commander-level personnel in the field or at the back office to assist in managing jobs. An incident commander could use this application to learn about an incident prior to their arrival at the scene of an incident, to monitor resources while at such a location, to request additional resources, to facilitate staff training, and to review incidents after their completion.

2.1.1. Option 1

This option had the benefit of a clear path to a well-defined scope, reducing the risk of an overly ambitious project plan which could not be executed in its entirety. With this choice the type of application (mobile or desktop) is obvious (it should be mobile), and the primary use case is naturally limited and well defined (given a reference to a vehicle the application should clearly identify any assets which are missing from that vehicle). However, this option had considerable overlap with one of the subsections of the ECE team's project, which involved creating an application for a dashboard-mounted Android device in an emergency vehicle which would provide functionality similar to that which we described above.

2.1.2. Option 2

This option would only have been practically useful if it supported use by personnel in the inhospitable conditions which follow an emergency such as a fire. Designing such an application would be challenging, highly subjective, and extensive user testing would be required before any authoritative claim could be made about the application's suitability for use under such conditions. The potential for the project scope to expand made this option less desirable.

2.1.3. Option 3

While the definition of the project scope for option 3 was less obvious than for option 1, we felt that this option had the potential to provide more value to the project stakeholders. The ECE team was not working on functionality of this nature, and we felt that it would provide a compelling demonstration of the potential of the technology being developed. Option 3 is also well placed to replace a number of processes which are currently performed manually.

2.2. Choice of Platform

We considered a number of platforms for the project. These included a Java desktop application, a web application, and iOS and Android mobile applications. We chose to implement an iOS application in Swift for primarily educational reasons. Having worked for 18 consecutive months on Java desktop applications the author of this report felt that there was more educational value to be obtained from working with a different set of technologies. Moving from desktop to mobile development also offered opportunities for personal growth and the development of technical skills.

2.3. Risk Management

After selecting a specific approach for the project we performed risk analysis, the results of which are listed below. Here we identify a number of the risks which were associated with the project as well as strategies for managing them.

- There are many aspects, complications and avenues for development related to this project. Thus its scope could have expanded beyond a manageable size.
  - We invested effort into defining the scope of the project. See Section 2.4. below.

- It is difficult to arrange an interview with someone at the incident commander level (the most likely user for this application). Thus there could have been factors which were not accounted for.
  - We arranged interviews with volunteer and professional firefights who, while not incident commanders themselves, were able to offer some insight into the workflows used by incident commanders.
  - Research recently carried out on behalf of Tait investigated the workflow of a rural fire incident and included information about the role of incident commanders. We studied this report and did not note anything which could have caused problems for the project.

- An application of this sort would likely be used in stressful situations, increasing the importance of rigorous user testing. This would require a significant commitment of resources to adequately ensure safe operation under such circumstances.
  - The aim of the project was defined as being the demonstration of the value of the proposed system. This did not require us to conduct exhaustive user testing to ensure that the application was tolerant of user error under stress. Thus, we did not concern ourselves with this aspect of the system.

- Without a background in the emergency services or extensive research of common practices and processes it is impossible to confirm that some of what this project aims to accomplish has not already been implemented.
  - A report produced for Tait included a section on existing software and made a number of proposals for new software which could be developed to help the rural fire service. According to this report, none of the most valuable features of the application proposed in this document had already been implemented, and in fact a number of our planned features were identified as being good candidate functionality for future projects.

- Given the sensitive nature of some of the work performed by the emergency services there may be legal requirements which this project should take into account.
  - We investigated any potential legal obligations, and spoke to our industry contact at Tait as well.
  - It was not the purpose of this project to produce an application which could be immediately deployed for use by the emergency services. Rather, we sought to produce a compelling product which demonstrated the utility of such a system.

2.4. Scope

There are many directions in which this project could have been developed. Without careful definition of the project's goals we would have risked its success. Here we list the areas which we chose to pursue in the course of this project and the areas which we did not.

- Target workflow
  - There are three stages to an incident: the callout, work at the scene, and work after the incident. To clearly define the scope of this project we concerned ourselves primarily with the second of these stages (work at the scene). While some work was related to the other stages our focus remained on the scene itself.

- Automatic analysis
  - The application will have access to enough data to perform some analysis and potentially suggest certain courses of action (eg: that a particular fire engine be assigned to a particular incident, given a number of event parameters). While we aimed to eventually implement some level of intelligent analysis we considered this strictly as a secondary goal. We intended to implement such a feature if time permitted.

- User documentation
  - We determined that this would be particularly important due to the nature of the work carried out by the emergency services. We do not intend for our application to be deployed without post-project development as so we did not attempt to produce complete user documentation. However, we produced enough documentation to address basic questions about the application (see Appendix 5).

- Security (eg: authentication and access control)
  - The data collected and accessed by the application may be sensitive in nature and there may even be legal requirements for its storage and retrieval. We consider these issues to be outside the scope of the project and chose not to address them.

- Reliability (data storage)
  - We did not concern ourselves with reliable data storage. Issues such as database consistency, robustness and redundancy were not relevant to this project.

- Data collection
  - This project was not concerned with collecting data from vehicles. This was the responsibility of the ECE project group. Any data which was required and which could not be provided by the ECE project group was fabricated/simulated, where it was reasonable to assume that such data could be available in a future iteration of the product.

3. Design

3.1. Low Fidelity Prototypes

We produced a number of low-fidelity prototypes to help with the design process and to facilitate further discussion with stakeholders. Two sample scenes are shown below. These represent views the user might be presented with while using the application.

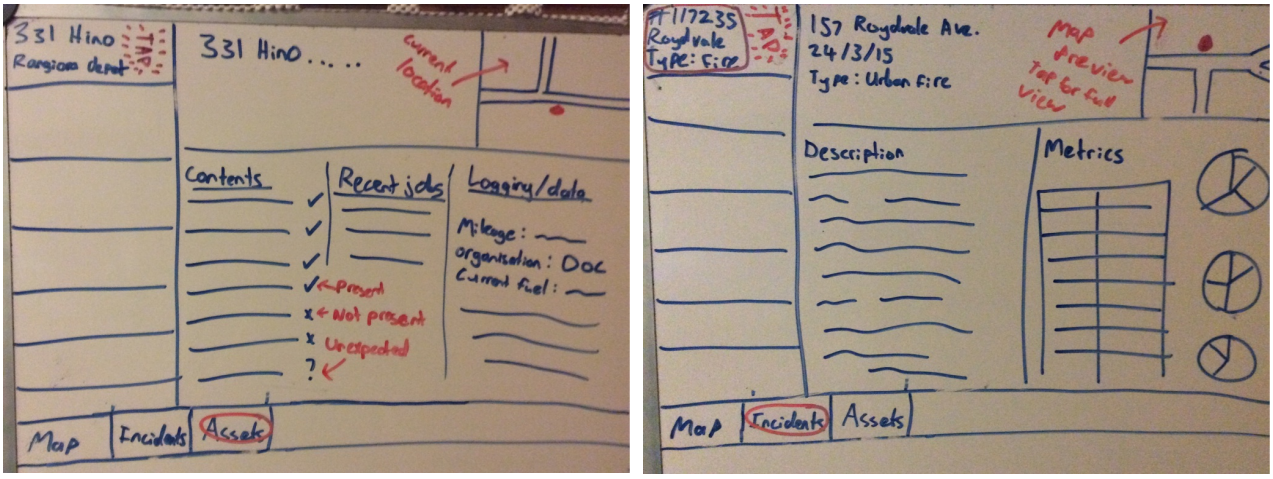


Figure 1. Sample of low-fidelity prototypes of the proposed application produced on a whiteboard

From this process we identified a number of important design considerations:

- Respecting operating system conventions makes the application more intuitive for the user, makes the application appear more professional (which in turn will makes the stakeholders more confident in the outcome of the project), and lowers development and maintenance costs (as standard elements can be used).
- Interaction cost should be minimised. We needed to consider which information was most important to the user, and at what times (and under what circumstances) they would want to see it. We needed to identify frequently accessed information and "pain points" early on in the design process.
- User workflows should be determined. This process was assisted by referring to the confidential report produced for Tait Communications which analysed existing processes in the emergency services.
- To make the application intuitive it was important provide interface cues and consider questions such as: "Is it natural to look for this over there?"
- We needed to balance the tension between providing all of the desired features and interface clutter.

3.2. High Fidelity Prototype

After completing the low fidelity prototypes described in Section 3.1 above we solicited feedback from our stakeholders and then began work on a high fidelity prototype. This was implemented in Swift and could be executed on a physical device.

The UI designs included in Section 3.1 above formed the basis of the interface we implemented in the prototype application. The only significant change we made was to make the map view always visible in the main part of the interface and to remove the global tab bar in favour of a smaller segmented control in the top left of the application interface. We did this because the map view is central to the user's motivation for using the application. It provides context and allows users to conveniently interact with geographically clustered items, rather than relying exclusively on an alphabetically sorted list. We also felt that the global tab bar took up an unnecessary amount of space when only two or three items could be identified to populate it.

Figure 2. shows the storyboard we created for the prototype application. The diagram is simple in appearance but involved extensive configuration and represents a substantial amount of effort spent learning about iOS technologies and frameworks. This investment will improved the efficiency of our development processes in subsequent stages of implementation.

We believe that the interface as implemented is extensible and usable, and will both support future development and assist users as they engage with the program. For example, new sections can be added to the right panel of the main view by adding items to the segmented control (although we expect that ultimately no more than three or four such sections will be needed).

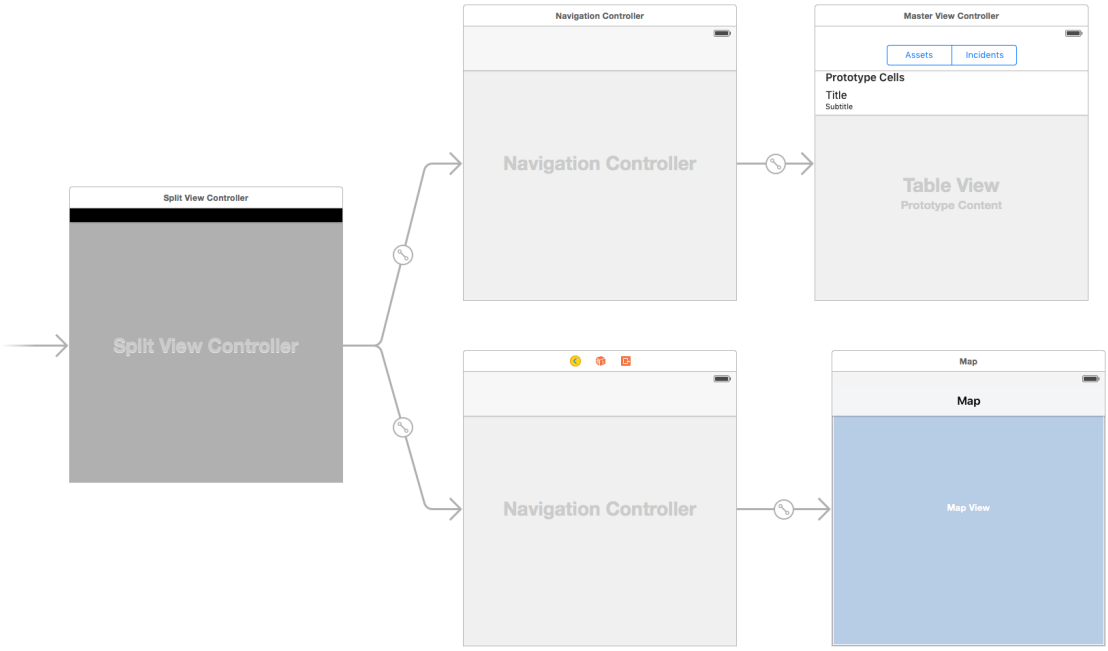


Figure 2. Storyboard for prototype application

As described in Progress Report 1 [2] we performed extensive requirements analysis and produced a number of prototype user interface designs. From this work we produced a simple prototype application (see screenshot in Figure 3. below) which could be run on the iOS simulator provided by Apple as part of their developer tools package or on a compatible iOS device. The prototype displayed only hard-coded data (assets and incidents) in a list view and a linked map view. Selecting an asset or an incident centred the map on it and displayed its callout. The map could be replaced with a detailed view of information related to an incident or asset by tapping the “i” symbol next to it.

This prototype represented a significant investment of effort towards gaining experience with iOS, Swift and the technologies associated with them. We also invested effort into learning about iOS programming best practices. This involved reading considerable amounts of documentation, attempting various implementations, discussing design and implementation methodologies with an experienced iOS developer, watching recordings of presentations given by Apple engineers, following online tutorials, and carrying out self-assigned exercises. We also used this prototype to solicit feedback from stakeholders to improve the implementation of the final design.

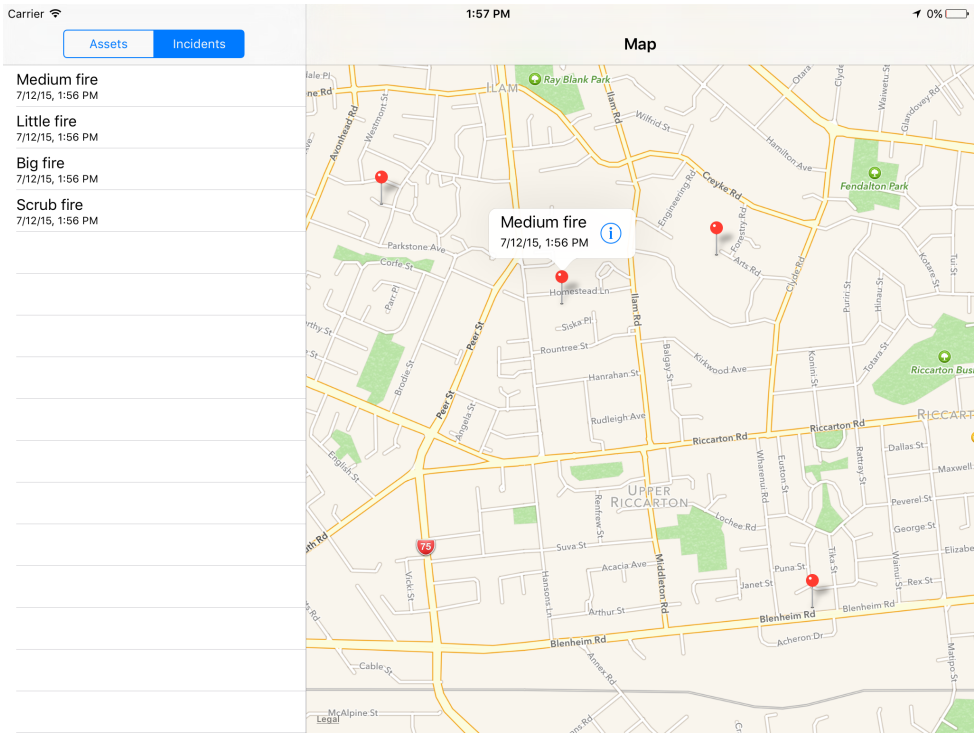


Figure 3. Screenshot of prototype application

4. Implementation

4.1. Data Access Layer

We originally conceived of using a data access layer (DAL) pattern to decouple our persistence management system from the higher level classes that used it. The DAL was intended to provide high-level methods for retrieving information which could be reimplemented should the persistence management system need to be modified. We chose Core Data as our persistence management system and thus placed the DAL above it. We created several iterations of a custom DAL for our application before discovering that Core Data offered much of the functionality we had implemented itself. Fortunately this did not necessitate any great changes to our system's architecture. We simply replaced the DAL's functionality with Core Data's where appropriate. Core Data itself thus took on the role of the DAL in our application.

4.2. System Architecture

The application obtains the data it presents from a server maintained by the ECE team. Changes to the local data store propagate through Core Data to the rest of the application and ultimately the user interface via event-driven programming. The data is obtained via a RESTful web service and persisted to disk by Core Data, through a custom syncing process.

Figure 4. shows a simplified overview of the entire system, including our application and the ECE team's projects. An on-vehicle server is responsible for collecting asset (and potentially incident) information and transmitting it to a centralised back-office server (both of which are to be developed and maintained primarily by the ECE team). The central server will maintain a core database which is the ultimate source of data for the system with which all other system components will synchronise.

The mobile application obtains its data from the central server through an API (defined and implemented in cooperation with the ECE team).

To assist in the testing and demonstration of our application we proposed a resource simulator. This was to be a process executed on the central server which modified data in a secondary database (the "simulated" database) as a reasonable approximation of real-world operation. However, during development we determined that it would be unfeasible to produce a real time syncing engine and, in consultation with our stakeholders, modified our approach. Instead we opted for a simpler manual syncing feature and added an additional endpoint to the API which allowed us to manually update the location of vehicles, allowing us to demonstrate the syncing functionality.

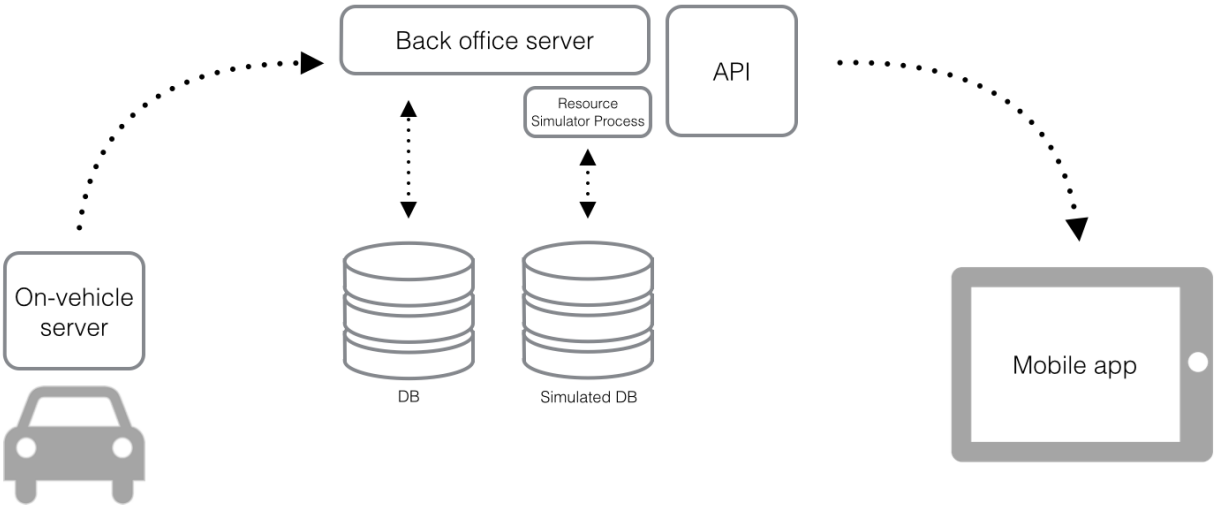


Figure 4. Application ecosystem, including data collection and retrieval

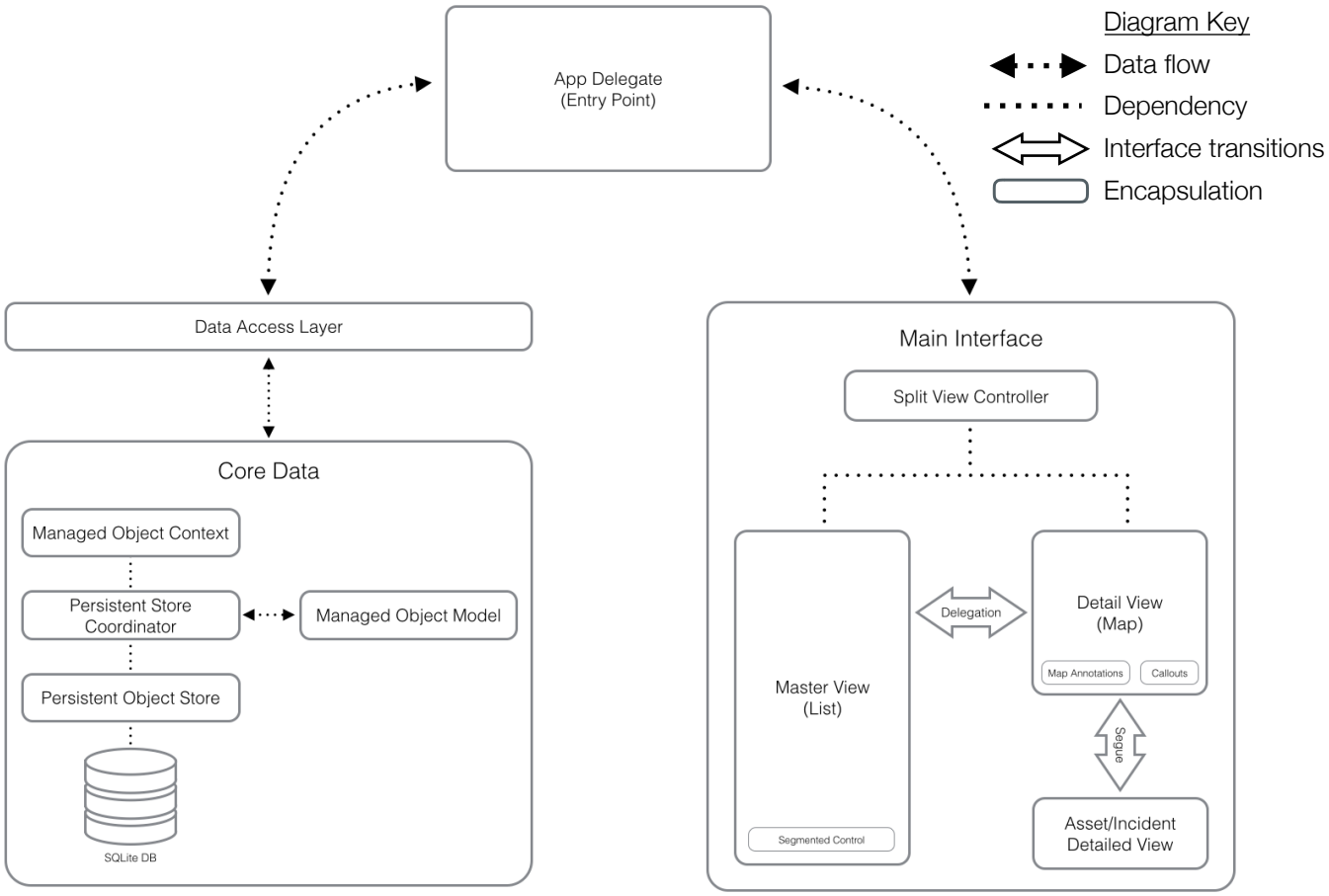


Figure 5. Application system architecture

4.2.1. Data Model[2]

Core Data provides a number of important components. An external data store (in our case an SQLite database) saves records to disk and guarantees persistence of data. A persistent object store handles mapping from records in the data store to objects in the application. The persistent store coordinator maintains a collection of data stores, and the managed object model represents the distinct entities within the store (these map roughly to tables in a database). The managed object context is responsible for maintaining an internally-consistent snapshot of a collection of managed objects in an application.

Core Data abstracts much of the complexity of persisting an object-relational model to disk, but it still requires a number of operations to retrieve data from the store. It also integrates well with the event-driven approach we have selected for propagating changes in the model to the interface.

4.2.2. Application Delegate

Delegation is a system by which components of iOS applications are able to pass responsibility for handling a certain event to one another. For example, a user may tap a button which delegates to a text field which responds by incrementing a counter. The application delegate is an object which handles notifications sent by the main application process. These notifications correspond to certain states of the application's execution, eg: the application has just loaded, the application has entered the background, the application has entered the foreground, the application has been terminated, etc. Thus the AppDelegate class is usually the focal point of iOS application development.

[2] Core Data information adapted from Apple developer documentation [5]

4.2.3. Application Interface

The main interface is presented as a split view and maintained by an instance of the UISplitViewController class. Split views are common in iOS applications and consist of two panes. The lefthand pane contains a list or table of some sort (the master view) which allows the user to select items which are then displayed in greater detail in the larger view on the right (the detail view). In our case the master view contains a list of incidents, assets and vehicles and the detail view contains a map which displays these items according to their geographical coordinates. A segmented control at the top of the master view allows users to toggle between displaying assets, incidents and vehicles in the master and detail views. Selecting an item in the master view pans and zooms the map to clearly display it, while tapping on an "i" button next to an item initiates a transition (termed a segue in iOS parlance) to a second view containing more detailed information about that item.

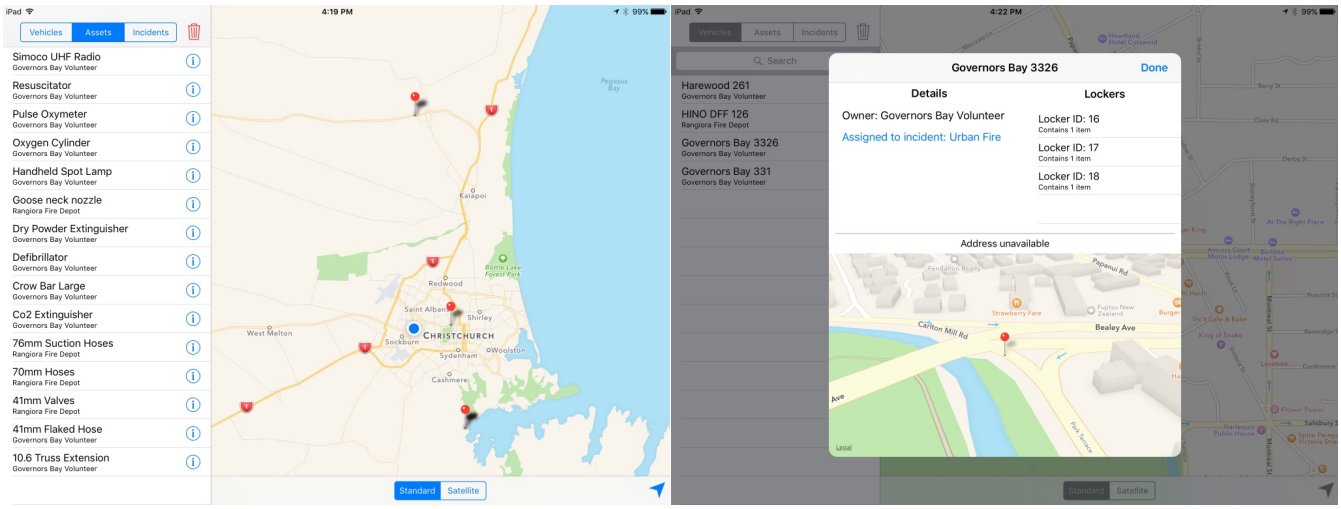


Figure 6. From left: Screenshot of application's main screen and screenshot of a detail view in the application

4.3. Development of Syncing Procedure

We progressed through three major revisions of the API, each of which focussed on specific areas of improvement. Unfortunately version 3 of the API was never fully implemented due to factors outside of our control. We provide some additional detail about this issue in Section 7.3.

4.3.1. Version 1 Goals

This version of the API was intended to expose a subset of the available information to allow experimentation at both ends of the connection. As development proceeded feedback was be accumulated and used to draft the specification for the next version of the API. Efficiency was not a concern. Only a lightweight syncing implementation was be created on the client as much change to the API was anticipated.

4.3.1. Version 2 Goals

API v1 was a rough draft. There were inconsistencies and some information was missing. API v2 was an iterative improvement and aimed fix some of the issues noted with API v1. API v2 also represented the first stable iteration of the API, allowing for greater investment in syncing functionality.

4.3.1. Version 3 Goals

API v2 was the first stable iteration of the API, but there were still issues and inconsistencies. API v3 aimed to resolve all of those issues. API v3 also aimed to improve efficiency by implementing a syncing system with hash codes. This would have allowed the client and server to determine which records needed to updated. Batching was intended to be used to avoid repeated HTTP requests.

4.5. Source Code

The final version of our application contained a little over 2,400 lines of code, with a further 1,000 lines of XML-style code having been generated by Interface Builder for defining the application's views. We produced a little more than 50 test cases, amounting to over 800 lines of test code. We invested significant effort into documenting our source code and are happy with the results. Almost every method and module was formally documented with concise descriptions of purpose and functionality, parameter and return value explanations, and usage examples. While some parts of the source code (particularly in the more complex sections of the syncing code) would benefit from refactoring, it is our opinion that our source code is of a high standard and, having applied best practices such as the open/closed principle, believe that it would be a suitable starting point for future development projects.

5. Testing

5.1. Testing in the Preliminary Stages of the Project

Before starting the project we had no prior experience of iOS development, the Swift language, or most of the technologies needed to produce the application. Creating the prototypes described in previous sections of this report helped greatly with this, but required a great deal of refactoring and trial and error. As a result we determined that it would have been infeasible to concurrently conduct automated testing, the frameworks for which were also unfamiliar to us on this platform. We used the first prototype of the application as the subject of a testing exercise once it was complete. This provided us with real-world code (which we were also familiar with) to test in what was then an unfamiliar environment. This had the added benefit of maximising the utility of the prototype, which in the real world would benefit stakeholders.

5.2. Testing in the Later Stages of the Project

We employed test driven development wherever possible. It is our experience that some implementation details are too difficult to test with TDD, and in these cases we used other approaches (eg: manual testing). One example of an area which was too expensive (in terms of effort) to test entirely using TDD was the syncing code. There are many places where things can go wrong during any synchronisation operation (eg: internal server errors, network errors, data read or write errors, data merging conflicts etc.) Thus for the test code to thoroughly test our synchronisation modules it would have needed to be highly complex. We deemed this investment of effort to be unnecessary and instead used TDD to test the core functionality of the modules and wrote some additional tests to catch some potential errors after the code was completed.

We conducted frequent manual testing of our application at all stages of development. We also performed regression testing after making major changes to the application. Between these two forms of testing we identified many errors in our code and produced an equivalent amount of additional test code to prevent those errors from recurring. It is difficult to produce code coverage analysis on Swift projects as many tools are yet to offer complete support for the language. Xcode reports our code coverage as being a rather low 27%, but this includes parts of the code base which we deliberately did not test, such as code directly related to the user interface. We produced a total of 52 unit test cases.

5.3. Automated GUI Testing

Xcode provides advanced support for automated user interface testing, but in our experience such technologies are difficult to use in practice. Because of this and the fact that Swift and its supporting technologies are still in development we chose not to adopt automated GUI testing for this project. We discussed this with our stakeholders and received their support for this decision.

5.4. Continuous Integration

While we considered setting up a continuous integration system we decided that it would not be suitable for this project. Continuous integration best serves teams of developers, rather than individual engineers. For example, with only one developer contributing code there was little risk of bad merges causing silent failures.

6. Tools and Technologies

6.1. Development Tools and Technologies

6.1.1. Xcode

Xcode is a proprietary IDE developed by Apple Inc. and provided to third party developers to help them develop applications for iOS and OS X. Xcode's first-party status gives it a competitive advantage which has discouraged most third party vendors from creating alternative iOS and OS X development environments. However, JetBrains' AppCode IDE has achieved some popularity, despite being unable to perform certain tasks (eg: interaction with Core Data, application archiving, and advanced storyboarding).

Fortunately Xcode offers many of the features considered essential in a modern IDE such as code completion, refactoring support, and inline documentation. Unfortunately not all of these features have been enabled for Swift programming. Despite this limitation it is our opinion that Xcode is still practical for developing Swift code for the purposes of this project and this is the IDE we used to develop the application.

6.1.2. Core Data

Core Data is a framework created by Apple which provides developers with high level operations for interacting with data serialised into XML, binary or SQLite stores. Xcode provides a graphical interface for managing these stores, further insulating developers from the low-level representation of their data. Core Data is a common method of data persistence in iOS applications.

6.1.3. Objective C

In our project proposal we suggested that Objective C would be a critical technology. After investigation we now believe that it will be mostly irrelevant to our objectives. Swift was designed as a replacement for Objective C for the purposes of writing software applications for Apple Inc.'s hardware devices. Swift applications can access all core iOS system libraries via traditional import statements, and Objective C modules can be used by Swift programs through the creation of bridging header files.

6.1.4. Interface Builder and UIKit

UIKit is an Apple-supplied framework which provides the infrastructure for maintaining the graphical user interface in an iOS application. In addition to offering a number of standard interface elements (eg: buttons, sliders, menu bars etc.) UIKit maintains event handling, deals with user input, and drives the main run loop of the program.

Interface Builder is a tool integrated into Apple's Xcode IDE. It allows programmers to build user interfaces by dragging and dropping graphical elements onto a digital canvas and configuring their positions with constraints. Such elements can also be linked to code by, for example, specifying a function to be called when a particular event occurs (eg: "double click", "touch up inside" etc.) This provides a convenient method for rapidly developing and visualising interfaces, but it does have some limitations.

Until recently it was impossible to place custom interface elements with Interface Builder. The removal of this limitation greatly enhances the utility of the tool, as many applications require graphical features which are not provided by UIKit.

It is possible to support devices with different screen resolutions by specifying relational layout constraints, but these are naturally more complex than the absolute references which can be used for applications which only need to run at a single resolution. Absolute references allow programmers to easily specify any location for a UI element (eg: x=10px, y=20px), while relational constraints mandate concepts such as “snap to centre”. While testing the limitations of Interface Builder we noted that it was simple to create a horizontal row of three buttons by snapping one to the left edge, one to the right edge, and one to the centre of the view. However, doing the same with four buttons was much more complicated due to the lack of a constraint which enforces the equidistant spacing needed to neatly arrange such a set of elements.

While relational layouts are harder to implement, Xcode provides a tool for automatically applying constraints to interfaces created in Interface Builder, greatly simplifying the process of creating responsive and functionality layouts. Our application uses mostly relative constraints, but some absolute references were used too.

6.1.5. Swift

Swift was Apple’s response to a number of problems perceived in its existing Objective C development pipeline. Swift provides a modern syntax closer to that of a language like Python which uses techniques like type inference and minimised boilerplate to increase programmer efficiency. While the flexibility of a language like Python comes at the cost of performance, Apple claims that optimised Swift code can run up to 40% faster than equivalent Objective C code. Writing test code to verify this claim requires detailed knowledge of the Swift and Objective C compilers, but anecdotal evidence suggests that Swift is indeed the more efficient language.

The Swift runtime and language specification were released in beta form in June 2014. In September of the same year Swift was officially released as version number 1.0. It is still considered under development, however, with compiler and language optimisations being released regularly. The volatility of the language is cause for concern to early adopters and implementers who may invest considerable effort into Swift programs which may be made redundant at a later date. To mitigate these concerns Xcode embeds a small Swift runtime library into every application allowing future runtimes to support legacy Swift code. Apple expects the Swift language to become stable at some point during 2015 or 2016. [8]

Swift’s modern syntax improves code readability and protects against common coding mistakes. For example, Objective C (like many languages) requires programmers to enter semicolons to delineate lines. In addition to requiring many redundant keypresses, semicolons are easy to forget, resulting in IDE or compiler syntax errors. Swift mimics Python by allowing programmers to forgo the use of semicolons. Another example is the now infamous “goto fail” SSL security vulnerability in Apple’s OS X and iOS operating systems. [9] This was caused by the unintentional duplication of a single line of code, as shown below.

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParam
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

Figure 7. Sample source code used in now-deprecated versions of OS X and iOS

“If” blocks which contain more than one line of code need to be wrapped in braces (eg: “{“ and “}”) in Objective C, but blocks which contain only a single line of code do not have this requirement. Had the language not allowed this shorthand it is highly likely that this vulnerability would never have been deployed. “If” blocks in Swift are delimited by indentation.

Swift also supports type inference, immutable constants and unicode. These further enhance code readability, increasing programmer efficiency.

Table 2. Language-level Technologies

Technology	Description	Experience Gained
	iOS Concepts and Technologies	
Delegation	A system whereby one object may delegate handling of an event to another object (NB: This is distinct from the observer pattern).	• Implemented delegates for a number of objects/classes.
Storyboarding	A technique for creating user interfaces by representing designs graphically to simplify modification and understanding. Xcode has a built-in storyboard feature which allows programmers to graphically create interfaces with drag and drop tools, as well as sophisticated configuration options and code integration (eg: for specifying code to execute when a particular interface event occurs).	• Created interface for prototype application using Xcode's storyboard feature.
Quickhelp	An Xcode feature which allows programmers to display code documentation inline by tapping on functions, variables, classes, protocols and other language constructs.	• Added documentation for classes and functions. • Formatted documentation for Quickhelp compatibility to allow inline viewing throughout Xcode.
Modal view controllers	A view controller which is presented on top of another, preventing any interaction with the obscured controller and its views.	• Implemented a number of different view controllers and presented them modally using programmatic cues. • Experimented with a range of methods for presenting modal view controllers and for controlling their appearance.
Popover view controllers	A view controller which is presented on top of another. Attempting to interact with an obscured view causes the popover view controller to be dismissed. Popover view controllers are typically rendered as small windows tethered to a point on the screen.	• Implemented popover view controllers as part of an experiment with a prototype of the application. • Compared the suitability of popover view controllers for presenting detailed information about assets to that of modal view controllers.
Segues	A transition between two views which is often animated.	• Used Interface Builder to define segues for a number of views. • Programmatically defined and initiated segues in view controller code.

Technology	Description	Experience Gained
	Frameworks and Classes	
Core Data	An Apple framework used for handling data and object persistence. Xcode provides a graphical interface for setting up and managing Core Data data models.	• Model editor • NSManagedSubclass • Data access layer
Quickhelp	An Xcode feature which allows programmers to display code documentation inline by tapping on functions, variables, classes, protocols and other language constructs.	• Added documentation for classes and functions. • Formatted documentation for Quickhelp compatibility to allow inline viewing throughout Xcode.
Core Location	An Apple framework used to obtain location information and to manipulate location data.	• CLLocationCoordinate2D • CLLocationDegrees • Location managers
MapKit	An Apple framework used for presenting maps in windows views. It provides extensive options for adding annotations, overlays and other graphics to maps.	• Added and configured a map view using a storyboard. • Implemented displaying user location (including obtaining system permissions). • Added annotations to map. • Implemented callouts for map annotations and added accessories to callouts (eg: “i” button) • Implemented MapViewDelegate.
NSNotificationCenter	An Apple framework used to broadcast event notifications to interested parties registered within the same application.	• Registering observers • Posting notifications • Adding user data to notifications • Retrieving user data
UITableView	Table views allow apps to efficiently display data in a tabular format.	• Configuring table views in an Xcode storyboard • Registering table view delegates • Repopulating table views
UISplitViewController	A view controller frequently used in iOS apps for dividing the available space in an app into a master (list) view and a larger detail view.	• Delegating from master view to detail view • Customising detail views
UITabBarController	A view controller used to present the user with a number of tabs along the bottom of the screen which assist in moving between a collection of views.	• Configuring tab bar controllers in an Xcode storyboard • Attempting to integrate UITabBarController into UISplitViewController.

Technology	Description	Experience Gained
	Swift Language Features	
Swift enumerations	Enumerations as found in most popular programming languages	• Applying Swift enumeration syntax in prototype app.
Generic typing in Swift	Generic typing, as found in some popular programming languages, eg: Java	• Applying generic typing in prototype application • Developing flexible functions for efficient code reuse
Closures	A language feature I was not previously familiar with - Essentially a programmatic scope. In Swift functions are a special case of closures.	• Implementing a callback • Using Swift's syntax for trailing closures
Protocols	A language feature allowing programmers to define abstract definitions which concrete classes can implement. These are similar in nature to Java's Interfaces.	• Swift is a highly protocol-oriented language [5]. • Worked with many built-in protocols, providing concrete implementations. • Defined new protocols.
Extensions	A language feature allowing programmers to add concrete functionality to any existing protocol class, without modifying the original.	• Made use of a range of built-in extensions. • Used extensions to separate Code Data managed properties from model classes, simplifying the process of modifying these properties. • Used an extension to add functionality to Core Data (a convenience method for creating instances of model classes).
	Miscellaneous	
Data Access Layer	A programming paradigm for separating data storage and retrieval logic from the main application.	• Implementing the singleton pattern in Swift • Implementing DAL in prototype • iOS data model best practices

6.2. Project Management

We organised and maintained this project using a variant of the Scrum methodology. We used the online tool Trello [7] to visualise and plan progress on the project. See Figure 8. below for a screenshot of our Trello board as it appeared partway through the project.

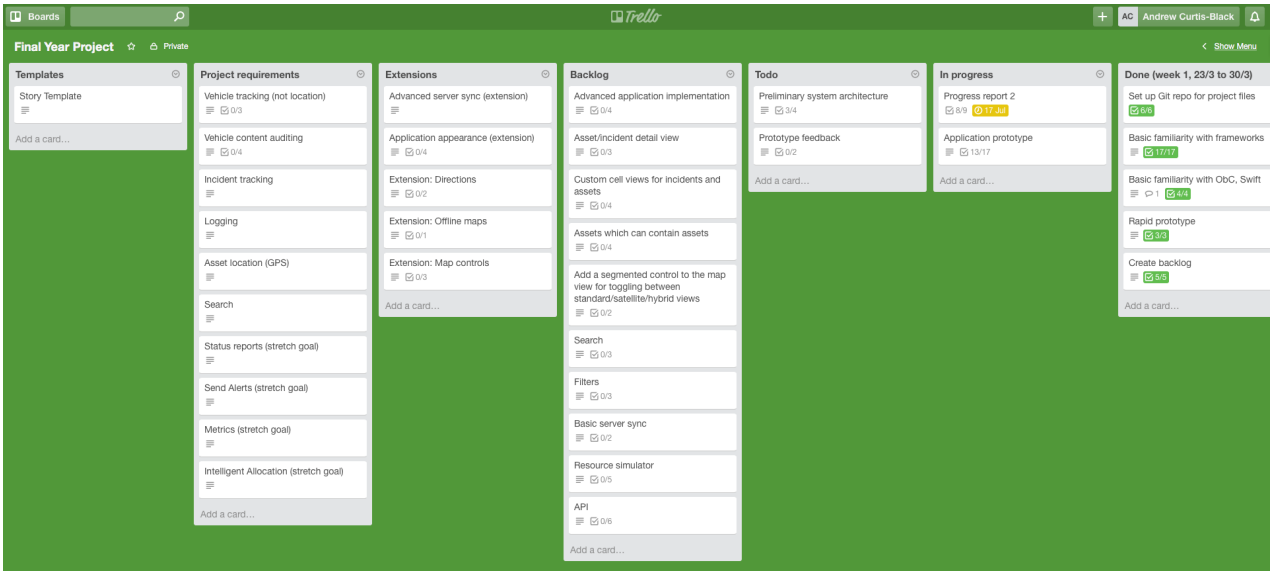


Figure 8. Screenshot of Trello board for this project

Our Trello board provided a clear visualisation of our approach to project management and supported it throughout the year. There were six main lists: Project Requirements, Extensions, Backlog, Todo, In Progress, and Done. The Project Requirements list did not contain feature stories, but rather the prioritised set of project requirements, as defined in the planning stages of the project. Each requirement had a checklist of criteria associated with it, as well as a concise explanation, simplifying the process of focussing our development efforts on work which returned value to the stakeholders. The backlog list contained a prioritised list of feature stories and served an identical purpose to the backlog used in Scrum projects. The Extensions list was a supplementary part of the backlog, containing those stories which related to non-critical project requirements (as classified in the project proposal [2]).

We did not institute a sprint cycle, as used in the Scrum methodology. Instead we divided our efforts into logical periods of work, each of which culminated in a major milestone. Such milestones included the due dates for the project proposal, progress reports, UC CSSE Postgrad Conference, and project demonstrations. At the beginning of each of these periods of work we shifted an appropriate number of tasks from the backlog into the Todo list, indicating our intention to complete all of these tasks in that period of work. The In Progress list was used to contain tasks which we were actively working on, and never contained more than two or three such tasks. Once all of the acceptance criteria for a task were complete it was moved to the Done list.

We also kept a record of the time we spent on the project. An extract from this record is available in Appendix 6.

7. Evaluation

7.1. Evaluation of Project Plan

Little deviation from our project plan (see project proposal [2]) was necessary. Thus it is our impression that the plan was well designed and performed as intended. We chose to adopt an Agile-style approach to this project and this afforded us the flexibility to adapt to unexpected events throughout the year. Chief among these was the continued interruption of tutoring commitments. We did not expect these commitments to be as demanding of us as they were. Nor did we expect the interruptions they necessitated to be as consistent as they were. Thus, our main failure in the planning stages of the project was to anticipate these issues and include them in our risk analysis. Had we allowed a buffer for additional tutoring commitments we would have been able to better schedule work on this project and more accurately scope our designs.

Despite this we met all of the essential requirements of the project, although we did not achieve any of the extensions (informally known as “stretch goals”).

7.2. Product Evaluation

The principle purpose of this project was to produce a prototype capable of demonstrating the potential of a resource management application for the New Zealand rural fire service. At the time of writing we consider the prototype we have created to be fit for this purpose. Furthermore, we demonstrated the prototype to our industry contact at Tait Communications near the end of the project and received positive feedback. Thus the ‘client’ for this project is satisfied with the outcome.

7.3. Evaluation of Collaboration with the ECE Team

We found that we were able to maintain a good separation between our project and the ECE team’s projects. This allowed us all to maintain a degree of independence in our work, reducing the likelihood of unexpected and/or shared delays. Thus, the time we spent defining our projects at the beginning of the year was well spent. By design, this project interfaced with only one part of the ECE team’s system, and one person on the ECE team had control over all of the areas which could affect this project. This simplified our interactions and increased our collaborative efficiency.

Only one problem occurred and that came at the end of the project. Because the ECE team’s project end date was several weeks before the end date for this project, the ECE team’s priorities shifted away from this area, making it increasingly difficult for us to schedule server-side changes. As a result of this the third and final version of the API was never fully implemented, and some functionality intended for the final version of the app could not be included.

8. Discussion and Conclusions

8.1. Assessment of Project Objectives

The table below shows that we met all of the essential objectives we defined for the project, but none of the exertions (“stretch goals”). The objective related to logging data was retrospectively modified during the course of the project. Below we present a discussion of the reasons for this.

This objective was intended to facilitate the replacement of manual paper-based logging routinely performed by fire service personnel with automatic electronic systems. However it became apparent that the best place to perform this work was on the remote server responsible for synchronising with the mobile application we produced. Thus meeting this requirement would have required the ECE team to do additional work which was not directly related to their own projects. We determined that this was unreasonable, and that ultimately the process of automatically logging this sort of information was out of the scope of this project. This project is concerned exclusively with presenting information to a user of a mobile device (specifically, an iPad). Thus it would be natural for us to implement functionality for the viewing of logged data, but not for the logging of that data. This was discussed with the supervisor of this project and one of the supervisors of the ECE team’s projects. The consensus was to modify the objective, converting it from a primary objective into an extension to the project.

Table 3. Project Objectives and their Status

Objective	Motivation	Classifcation	Status
Allow users to view the contents of tracked vehicles.	Vehicle contents tracking has been identified as an important capability which is not provided by existing systems.	Essential	Complete
Present users with a list of incidents (past and present).	Incidents are an existing conceptual component of the workflow used by workers in the emergency services. Including this concept in our application will make it more intuitive, and also provide a convenient navigational tool.	Important	Complete
Automatically log data.	Existing manual processes are time consuming and imprecise.	Important	Objective modified
Present users with the locations of tracked vehicles.	This information is critical to users and is provided by existing systems.	Important	Complete
Allow users to search for incidents.	The program will track more incidents than can be practically searched manually.	Important	Complete
Generate status reports for incidents on demand.	Users may not have time to acquaint themselves with all the details of an incident. Status reports would provide users with a way of quickly appraising a situation, and of exporting information from the application.	Stretch goal	Incomplete
Be capable of sending alerts to tracked vehicles.	Bidirectional communication would empower users to respond to the observations they make with the application.	Stretch goal	Incomplete
Provide graphical and textual summaries of relevant metrics.	Data collected by the application will be most useful to users through real-time visualisations.	Stretch goal	Incomplete
Provide suggestions for resource allocation based on automatic analysis.	This will assist with resource allocation, which can be a highly complex task dependent on many factors.	Stretch goal	Incomplete

8.2. Assessment of Project Requirements

The status of the project objectives above is reflected in the status of the functional requirements for the project. This is expected, as the former were derived from the latter. We met all the necessary functional requirements for the project and we met all the non-functional requirements we defined. Thus we consider these aspects of the project to be successful.

Table 4. Assessment of Functional Requirements

#	Objective	Classification	Status	Comments
1	The application shall allow users at Firecom to view the contents of tracked vehicles.	Required	Requirement met	
2	The application shall present incident commanders with a list of incidents. It must be possible to view completed (past) incidents as well as active (current) ones.	Required	Requirement met	Past and present incidents are distinguished with a coloured badge which reads either “complete” or “incomplete”, as appropriate.
3	The application shall automatically log resource data, mirroring/replacing existing manual logging processes which are carried out by staff at fire depots. Examples: Distance travelled by vehicles, vehicles attending a job.	Required, modified to extension	Requirement not met	See discussion in Section 8.1 above.
4	The application shall present incident commanders with the locations of tracked assets.	Required	Requirement met	Due to limitations in the data provided by the ECE team’s system, assets’ locations are derived from the vehicles in which they are currently located.
5	Administration staff shall be able to search for tracked incidents with various parameters. Examples: Job number, requirements, equipment usage.	Required	Requirement met	
6	The application shall be capable of generating a status report for an active incident. This could be used by an incident manager to quickly appraise themselves of the situation prior to their arrival.	Extension	Requirement not met	

#	Objective	Classification	Status	Comments
7	It shall be possible to send a range of alerts/ messages to tracked vehicles. Example: “Reassigned to incident #111”	Extension	Requirement not met	
8	Incident commanders shall be able to view metrics for incidents, resources, and work units (e.g.: brigades). Examples: Total distance traveled by all vehicles, fuel consumption, total cost of incident.	Extension	Requirement not met	
9	The application shall be capable of suggesting which resources an incident commander should allocate to an incident (based on location, incident requirements, resource readiness, job urgency etc.)	Extension	Requirement not met	

Table 5. Assessment of Non-Functional Requirements

#	Objective	Status
1	The application shall be suitable for use by an incident commander (IC), as in the New Zealand Rural Fire Service. The IC may use the application to gather information about the incident prior to arriving at the scene, to monitor resources during an incident, to request additional resources, and/or to review an incident after it has finished.	Requirement met
2	The project aims to improve the operations of the clients, not to introduce software which hinders or complicates existing processes.	Requirement met
3	The application shall be a concept related to rural fire used to facilitate discussion for further feedback.	Requirement met
4	The application shall be self-contained. It shall not require extensive integration with any existing systems in order to add value or otherwise be judged successful.	Requirement met
5	The application shall run on an iPad and be produced using Swift, Cocoa Touch and/or Objective C	Requirement met
7	Though the application is intended for use in stressful situations, for the purposes of this project this shall not be a primary concern. Making the application robust to user error while under stress could require a prohibitive commitment of time and effort.	Requirement met

8.3. Learnings

Over the course of this project we have learned a number of things. These include abilities related to new technical skills, knowledge of specific frameworks and languages, task management, planning, problem solving, evaluation, risk analysis, and communication.

In Table 2 we identified a number of new technologies with which we familiarised ourselves during this project. These frameworks, language features and concepts will change with time, effectively decaying the value of our specific knowledge of them. However, the process of learning about these technologies and applying them in practice is an even more valuable skill and experience in and of itself, and this we shall retain. We note that one unexpected lesson was that learning how to use a mature framework is significantly harder than becoming familiar with a new language (at least once one has attained familiarity with a number of similar languages already). Frameworks can encapsulate incredible complexity which is often expressed in nuances and "gotchas" that can stall development for hours or even days. These sorts of delays should be factored into any project plan with generous safety margins.

Despite anticipating that the planned syncing functionality would require a great deal of effort to implement, and despite making allowances for this we still underestimated the time required to achieve this goal. There are many layers to syncing code including the server-side logic, the network stack and the operating systems of the entities sending and receiving the information. Thus errors can occur in many places, and for many reasons, necessitating a certain degree of complexity in even the simplest syncing solution. One specific problem we encountered was that the server and the client have to, to an extent, adopt parallel implementations for tasks such as hashing for the purposes of determining the equality of remote records. But when the client and the server do not even share a common programming language this can be complicated to achieve. In our case the server and the client used different encodings for strings, and additionally defaulted to representing zero as "0.0" and the other to "0" in string values.

The author of this report had significant commitments as a tutor throughout the year. We did not expect these commitments to require as great an investment of time as they did. As a result the effort available for allocation to this project was restricted in ways we did not anticipate at times we did not expect. It was not possible to foresee the extent of this risk, but we should have recognised that it represented an unknown quantity and allowed ourselves additional safety margins.

8.4. Future work

At the outset of the project we defined a number of extensions or “stretch goals” to be completed if time allowed. These form a natural starting point for future work on the program. Below we provide a number of suggested features for future versions of the application as well as an example user story for each.

Table 6. Future Work

Feature	Example User Story
Directions	As a user I want to be able to get directions to points of interest on the map (e.g.: assets, locations)
Offline maps (caching)	As a user I want to be able to make use of the app in situations where I may not have an internet connection.
Advanced map controls	As a user I want to be able to change the level of detail shown on the map. I may not always want to see callouts for landmarks and locations like malls and parks.
Status report generation	As an administration staff member I want to be able to quickly generate status reports about current and past incidents (reports would probably look different for current and past incidents). Reports for current (ongoing/active) incidents would be used to appraise staff of the situation as they arrived to work on the incident. Reports for past incidents could be used for training, to review processes, or to review asset usage.
Alerts	As an incident commander I want to be able to notify staff in vehicles about various events, and provide them with instructions. E.g.: You should not have that pump, it belongs to another vehicle.
Metrics	As an incident commander I want to be able to view metrics for incidents, resources and work units (brigades, crews). E.g.: Total distance traveling by a vehicle, fuel consumption, total cost of incident etc.
Intelligent resource allocation	As an incident commander I want the app to automatically suggest which assets should be allocated to an incident (based on location, requirements, readiness, urgency etc.)
Automatic logging	As a Firecom staff member I want the app to automatically log as much relevant data as possible, so that I do not have to.
Advanced syncing	As a user I want the application to automatically update itself with the latest data and display it in real time (without manual refreshes).

Appendix 1

3311 Equipment Checklist

GOVERNORS BAY 3311 EQUIPMENT INVENTORY CHECKLIST

Month of : ***January 2015***

#	Cab Front	OK	Fail
2	Traffic Wands		
1	Wajax Tool Kit		
1	Oxygen Cylinder & Gauge		
3	Rural Goggles		
2	Simoco UHF Radios		
1	Fuel Card		
1	Emergency Response Guide Book		
3	Torch		
1	CCC Fire Plan Folder		
1	Risk Plans Folder (Red)		
1	Road Map Book		
1	Box Medical Gloves		
3	High Vis Jerkins		
1	Jumper Leads		
1	Tube of Sunscreen		
1	Trauma Kit		
1	Blankets		
	Bottled Water		

#	Nearside Rear Locker (Lower)	OK	Fail
1	Brown Bros Portable Pump		
1	Portable Dam		
1	Rega Backpack Sprayer		

#	Nearside Bin	OK	Fail
4	100mm Suction Hose & Strainer		
2	Wajax Suction Hose & Strainer		
3	Wheel Chocks		
1	100mm Strainer & Basket		
1	Elephant Truck Monsoon Filler		
4	Suction Spanners		
2	Browns Bros Suction & Strainer		
1	Tanker Dump hose		
6	Road Cones		

#	Offside Front Locker	OK	Fail
3	Forestry Pack - Plexone nozzle & dry line		
	Forestry Couplings Asst		
2	41mm Dividing Breeching		
1	By-pass couplings		
1	Priming Pump		
2	41mm short hoses		
1	Wajax Tool Kit		

#	Offside Rear Locker (upper)	OK	Fail
2	Polaski Axe		
2	Slasher		
1	NZFS & Hot Zone Tape		
1	Wajax Backpack frame		
4	Hydroblender Capsules		
1	Branch - Fyrex		
1	Branch - Plexone on stantanious coupling		
1	Adaptor - Instantanious to forestry		
1	Boltcutters		
1	Dividing Breeching		

	Offside Rear Tray	OK	Fail
3	41mm Hose on Bight		
4	Hose 70mm Coiled		
1	Hose 45mm Coiled		

#	Rear Locker & Back Step	OK	Fail
1	Standpipe		
1	Hydrant Key & Bar		
1	Wajax Pump		
1	Portable Pump - Superjet		
1	Fuel Can - Petrol 2 Stroke		
1	Fuel Can - Petrol Unleaded		
1	Foam A Class 20ltr		
1	Short Feeder		
1	Hydroblender & 2 Capsules		
3	Shovels		
1	McLeod Rake		

Firefighter's Signature :

Special Notes:

Appendix 2

DFF 126 Checklist (Pg. 1)

APPLIANCE DRIVER/OPERATOR VEHICLE CHECK SHEET

Truck Km 1. _____ 2. _____ 3. _____ 4. _____ 5. _____ 6. _____

RUC due KM _____ *If Truck km is 500km or less than RUC due km buy more RUC*

Date COF Due _____ **Date Registration due** _____

Checks to be Carried Out	1. Date Checked	2. Date Checked	3. Date Checked	4. Date Checked	5. Date Checked	6. Date Checked
Mechanical & Visual checks	✔ ✘	✔ ✘	✔ ✘	✔ ✘	✔ ✘	✔ ✘
Fuel Level (min ¾ Tank)						
Engine Oil level o.k.						
Radiator Water Level						
Appliance Radio Switched Off						
All Interior Lights Working						
Exhaust & Air Brakes Operating correctly						
Vehicle & Pump Log Books (Filled In)						
Portable Radio & Head Set Operational						
Inside Appliance Clean & Tidy						
Headlights (Dip/Full)						
Grill Flashers						
Park Lights						
Indicators (Front & Rear)						
Brake Lights						
Number Plate Light						
Reversing Light						
Working Flood Lights (x 4)						
Hazard Lights						
Locker Lights						
Beacons (Front & Rear)						
Water Tank Level (At fire full, At depot empty)						
Pump Control Valves Operational						
Visual Tyre Check						
Load Security (Dam, Cones, Signs)						
Vehicle Body Work (Any Damage Report)						
Wajax Pump Re-commissioned & Cover locked down						
Waterous Pump Re-commissioned & Cover locked down						
Full wash of vehicle upon return to Fire Depot						
Full Inventory of Equipment Checked	***Replace any used equipment requiring servicing + replace Hose, Soap capsules etc***					

	Driver or Checking Persons Name (Please Print Name)	Signature	Comments
1.			
2.			
3.			
4.			
5.			
6.			

Appendix 3

Interview Summaries

Information from New Zealand Police

- "A simple example is if there is a callout to set up a road block. Comms will look to see which is the closest unit – does the car have spikes in the boot? Is the officer qualified to deploy them? Is there an appropriate weapon in the car if needed e.g. Glock or (I think) AK rifle or similar.
- I would see training / skills as an asset also, e.g. if comms need to dispatch someone to an attempted suicide, where is the closest officer with suicide negotiation training."

Nick Rayner (volunteer firefighter)

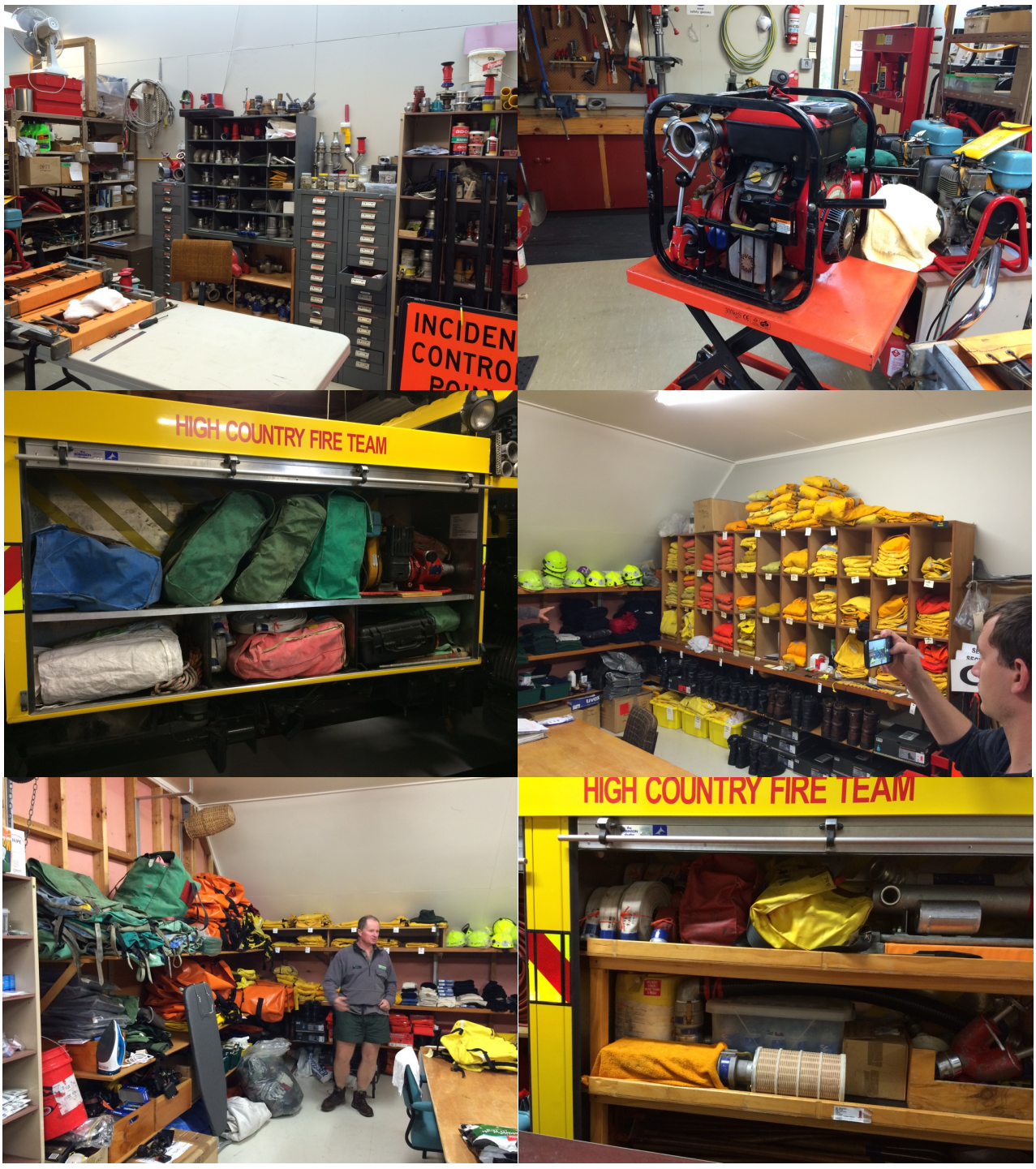
- Assets include:
  - Tanker 331 (3 pumps)
  - Support van (tarpaulins, first aid kit, generator)
  - Appliance/fire engine
  - Hydraulic cutters
  - Helicopter
  - Monsoon buckets
  - Westpac Helicopter
- Same gear taken to every job
  - If specialist gear needed, requested separately and transported separately (to brigade)
- Job workflow
  - 111 call
  - Firecom notify brigades in area
    - Brigades respond K0 if unable, K7 if attending
    - Firecom will stand down surplus brigades if multiple respond
  - At scene, people arrive at different times and take on different roles, as needed
    - Some high ranking officers may arrive and take command only if needed (if things are in hand, they will not take command)
  - Incident commander in charge at high level
  - After job, pack up
    - After jobs gear may be swapped between crews - no formal packup system
  - Log everything
    - Miles on speedo (compared to log entry from before appliance left)
    - Oxygen tank levels
    - Petrol usage
- Problems
  - After jobs gear may be swapped between crews - no formal packup system
  - Tracking when items need servicing
  - Sometimes council workers remove items from truck when doing maintenance (think item not needed, causes tension with brigade)
  - IC isn't automatically notified when brigades arrive at scene
- Observation
  - There are two levels of interest. High level incident control and low level brigade. High level wants information about scene and location of large assets (eg: appliances). Low level wants to know where specific items are located on vehicles, and general information about the fire.

Richard Parker (Scion rural fire research group project lead)

- Types of jobs
  - Vegetation fire
    - Grassland (up to knees)
    - Scrub (up to 2-3m high, gorse)
    - Forest
  - Rural fire do not deal with structures of vehicle fires
  - Crew don't get much info about fire prior to arriving at scene
- Additional equipment
  - "Dam" (like a large paddling pool) used to store water if no good water source nearby (pump water over long distances to dam).
- Problems
  - People have trouble locating equipment on truck sometimes. Would like to be able to find out where item is by asking app.
    - Equipment often not where you'd expect on truck (different place to the right labels)
    - Example: Lost nozzle. Couldn't find it for a long time. Would have been handy to confirm that it was definitely in the vehicle, so they knew it was worth continuing to look.
  - Sometimes equipment taken (thieves, other crews etc.). A "where is my pump" feature would be useful.
  - Large amount of paperwork. Automating information entry would be valuable.
    - Logging often done after the job when everyone tired and dirty (black from soot)

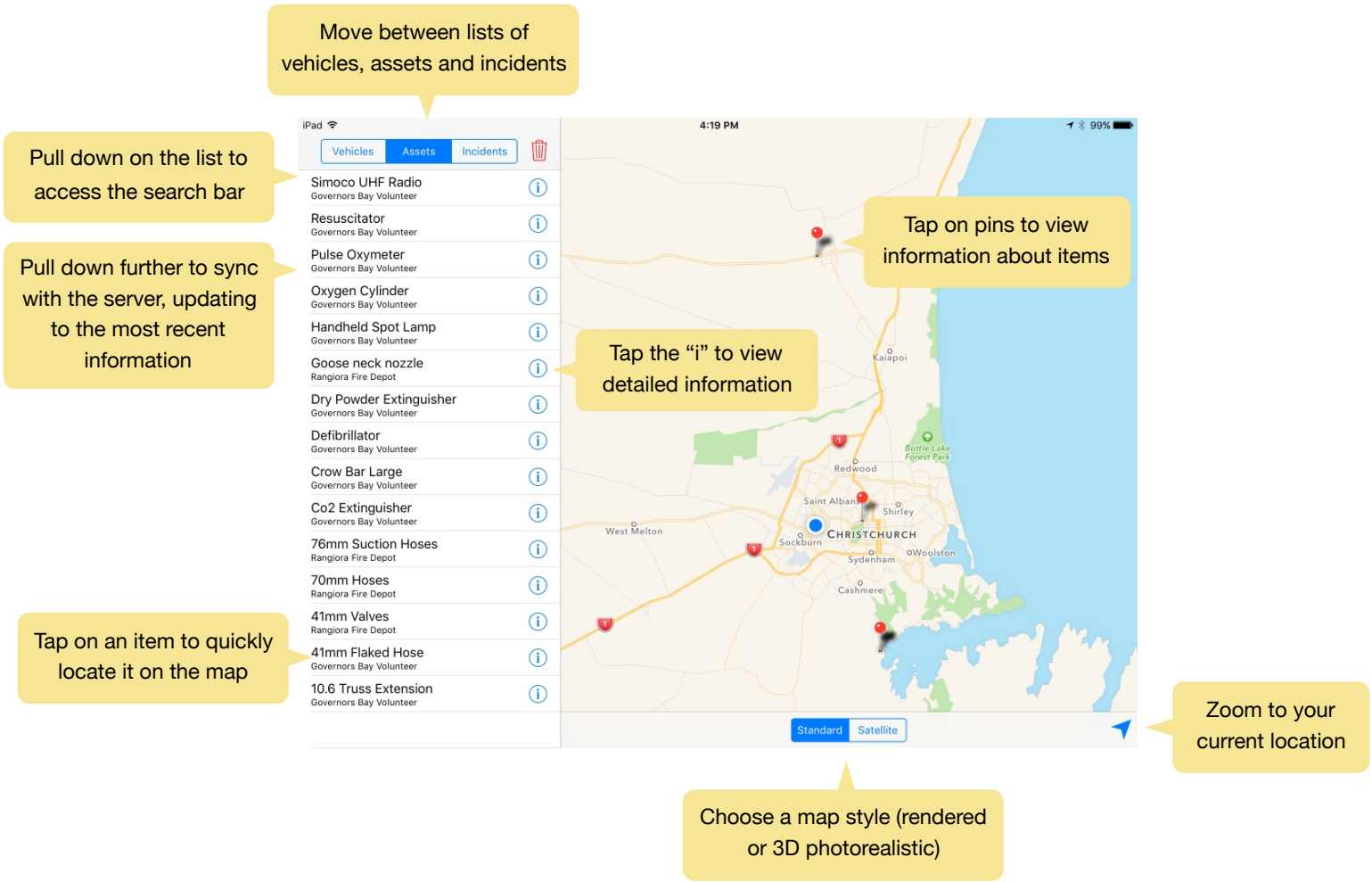
Appendix 4

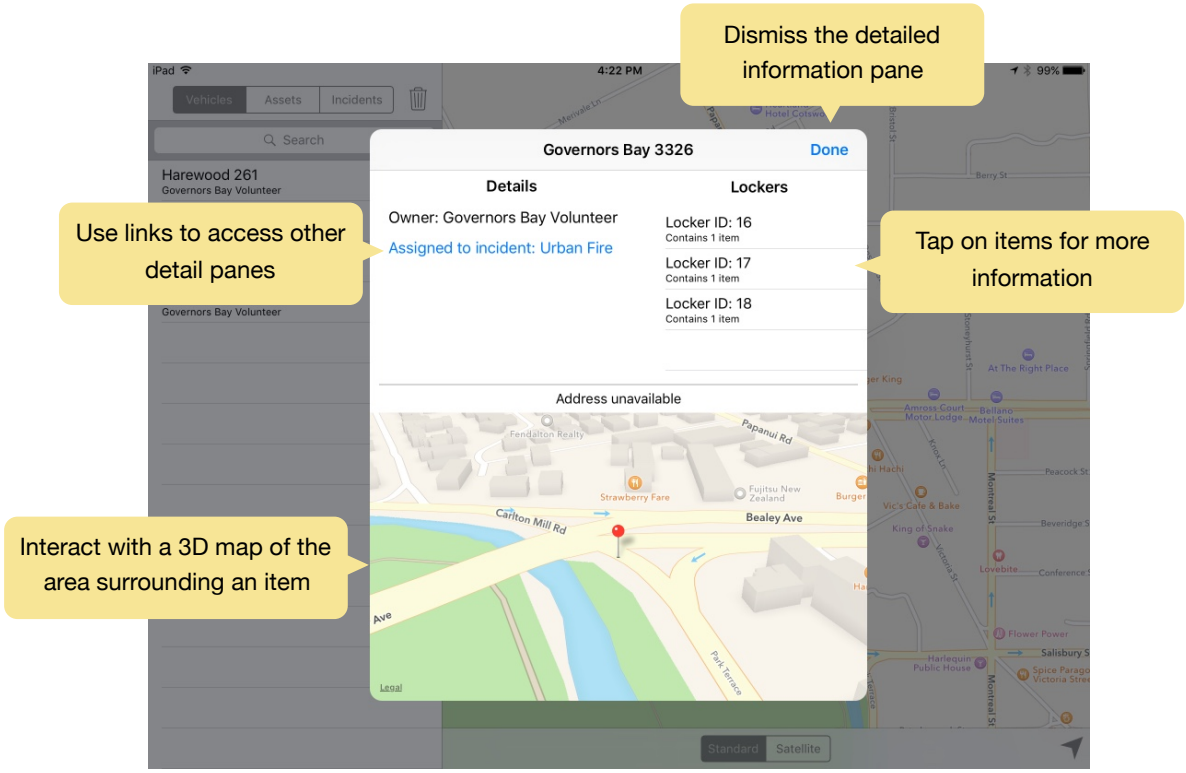
Selection of Photos from Rangiora Fire Depot Visit



Appendix 5

User Documentation




Dismiss the detailed information pane
Use links to access other detail panes
Tap on items for more information
Interact with a 3D map of the area surrounding an item
Governors Bay 3326
Done
Details
Lockers
Owner: Governors Bay Volunteer
Assigned to incident: Urban Fire
Locker ID: 16
Contains 1 item
Locker ID: 17
Contains 1 item
Locker ID: 18
Contains 1 item
Address unavailable
Vehicles
Assets
Incidents
Search
Harewood 261
Governors Bay Volunteer
Standard
Satellite

Appendix 6

API Specification (v1)

Endpoints

URI fragment: /vehicle/lockers/{plate_num}

Request type: HTTP GET

Description: Gets Locker objects from a given string of a plate number for a vehicle (can be a truck).

Parameters:

- plate_num: The alphanumeric licence plate number of the vehicle on which the desired lockers are stored.

URI fragment: /assets/{plate_num}

Request type: HTTP GET

Description: Obtains a sequence of Asset objects stored on a specific vehicle.

Parameters:

- plate_num: The alphanumeric license plate number of the vehicle which contains the desired assets.

URI fragment: /rfa/vehicles/{rfa_id}

Request type: HTTP GET

Description: Obtains a list of Vehicle objects belonging to a fire authority.

Parameters:

- rfa_id: The integer ID of the fire authority entity which has ownership of the desired vehicle.

API Specification (v2)

Endpoints

/fire_authorities

Request type: HTTP GET

Description: Obtain a list of fire authority IDs.

Parameters: None

URI fragment: /fire_authorities/{auth_id}

Request type: HTTP GET

Description: Obtain detailed information about a fire authority.

Parameters:

- auth_id: The ID of the desired fire authority.

URI fragment: /incidents

Request type: HTTP GET

Description: Obtain a list of incident IDs.

Parameters: None

URI fragment: /incidents/{incident_id}

Request type: HTTP GET

Description: Obtain detailed information about an incident

Parameters:

- incident_id: The ID of the desired incident.

URI fragment: /assets

Request type: HTTP GET

Description: Obtain a list of asset IDs.

Parameters: None

URI fragment: /assets/{asset_id}

Request type: HTTP GET

Description: Obtain detailed information about an asset.

Parameters:

- asset_id: The ID of the desired asset.

URI fragment: /vehicles

Request type: HTTP GET

Description: Obtain a list of vehicle IDs.

Parameters: None

URI fragment: /vehicles/{vehicle_id}

Request type: HTTP GET

Description: Obtain detailed information about a vehicle.

Parameters:

- vehicle_id: The ID of the desired vehicle

URI fragment: /lockers

Request type: HTTP GET

Description: Obtain a list of locker IDs.

Parameters: None

URI fragment: /lockers/{locker_id}

Request type: HTTP GET

Description: Obtain detailed information about a locker.

Parameters:

- locker_id: The ID of the desired locker.

API Specification (v3)

Syncing

A) Syncing Procedure

1. For each endpoint (/assets, /incidents/etc.) there should be a /sync option (eg: /assets/sync).
2. When the client makes a request using /sync it will supply JSON data in the following format: {id: hash, id: hash, id:hash … }. This JSON data would be attached to the /sync request, which would be performed as a POST request.
3. The server responds to a /{endpoint_name}/sync request by building up an array of data (henceforth referred to as "the array") to send back. This will be sent as JSON data in the following format: {added: [{…record_data…}], modified: [{…record_data…}], removed: [id_num, id_num...]} Where "{… record_data…}" is all the data associated with a particular record, and where "id_num" is only the ID for a given record in that table (because if a record has been removed the client does not need to know anything other than its ID).
4. Any IDs which are included in the client's /sync request and which are not also in the {endpoint_name} table in the DB are included in the "removed" array.
5. Any IDs which are not included in the client's /sync request and which are in the {endpoint_name} table in the DB are included in the "added" array.
6. Any IDs which are included in the client's /sync request and which are in the {endpoint_name} table in the DB are checked.
    6.1. The server computes a hash value for each such record in the {endpoint_name} table in the DB (see procedure for computing this hash value below)
    6.2. For each such record in the {endpoint_name} table in the DB the server compares the hash value included in the client's /sync request with the value the server just computed (see step above).
    6.3. When the hash values match the server takes no further action. When the hash values differ the server adds the record to the "modified" array.
7. The server then sends the JSON data to the client, which uses it to synchronise its local DB.

B) Record Hashing Procedure

1. Inputs: A single record from a DB table, a number M.
2. Append all fields in the record (ID, name, foreign keys etc.) into a single Unicode string.
3. For each character in the string compute that character's numeric value.
4. Sum these numeric values to give a total, T
5. Hash code = T % M

C) Additional notes about Hash algorithm

1. The client and server will agree on a value for M. The only condition is that M be substantially smaller than the majority of the T values which are generated, and that M is large relative to the number of records in any given table.
2. M will never change between computations, requests etc.
3. The goal here is to be able to tell when a record has been modified. We are not interested in knowing if a record referenced by another record has changed. For example, if the record is a Vehicle, that record will reference Locker records which will in turn reference Asset records. If a locker is removed from the vehicle then we should signal this change in the sync somehow. If an asset is removed from a locker on the vehicle we do not need to signal this change in the sync. If the name of an asset on a locker on the vehicle is changed then we do not need to signal this change in the sync. Thus, when building the string, we include the vehicle's own data (ID, name, location, locker IDs, etc.) but not "second level" data (the IDs of the assets in the locker) or anything deeper.

Endpoints

/fire_authorities

Request type: HTTP GET

Description: Obtain a list of fire authority IDs.

Parameters: None

URI fragment: /fire_authorities/{auth_id}

Request type: HTTP GET

Description: Obtain detailed information about a fire authority.

Parameters:

- auth_id: The ID of the desired fire authority.

URI fragment: /fire_authorities/sync

Request type: HTTP POST

Description: Request new fire authority data from the server in order to synchronise local and remote data stores.

POST data: See above ("syncing")

Parameters: None

URI fragment: /incidents

Request type: HTTP GET

Description: Obtain a list of incident IDs.

Parameters: None

URI fragment: /incidents/{incident_id}

Request type: HTTP GET

Description: Obtain detailed information about an incident

Parameters:

- incident_id: The ID of the desired incident.

URI fragment: /incidents/sync
Request type: HTTP POST
Description: Request new incident data from the server in order to synchronise local and remote data stores.
POST data: See above (“syncing”)
Parameters: None

URI fragment: /assets
Request type: HTTP GET
Description: Obtain a list of asset IDs.
Parameters: None

URI fragment: /assets/{asset_id}
Request type: HTTP GET
Description: Obtain detailed information about an asset.
Parameters:

- asset_id: The ID of the desired asset.

URI fragment: /assets/sync
Request type: HTTP POST
Description: Request new asset data from the server in order to synchronise local and remote data stores.
POST data: See above (“syncing”)
Parameters: None

URI fragment: /vehicles
Request type: HTTP GET
Description: Obtain a list of vehicle IDs.
Parameters: None

URI fragment: /vehicles/{vehicle_id}
Request type: HTTP GET
Description: Obtain detailed information about a vehicle.
Parameters:

- vehicle_id: The ID of the desired vehicle

URI fragment: /vehicles/sync
Request type: HTTP POST
Description: Request new vehicle data from the server in order to synchronise local and remote data stores.
POST data: See above (“syncing”)
Parameters: None

URI fragment: /lockers

Request type: HTTP GET

Description: Obtain a list of locker IDs.

Parameters: None

URI fragment: /lockers/{locker_id}

Request type: HTTP GET

Description: Obtain detailed information about a locker.

Parameters:

- locker_id: The ID of the desired locker.

URI fragment: /lockers/sync

Request type: HTTP POST

Description: Request new locker data from the server in order to synchronise local and remote data stores.

POST data: See above (“syncing”)

Parameters: None

Appendix 6

Extract from Project Time Log

Below is an extract from the time log spreadsheet we maintained during the project.

Date	Work performed	Comments	Time spent (approx)
Jan 20	Began discussing final year project options with Clive.	Clive provided a series of documents detailing iterative improvements to a group of loosely connected projects to be performed by summer interns from Tait.	0h 15m
Jan 30	Emailed Andreas about possibility of supervising project	Clive expressed interest in supporting a number of final year projects. To get a project together need a supervisor. Andreas has worked with Tait/ Clive in the past and has lectured me twice. Good fit for all parties.	5m
Feb 10	Meeting with Andreas to discuss project	Andreas agreed to supervise project. Warned that his area is networking not software engineering, but said he was still happy to help. I explained the project to Andreas. Andreas suggested that more should be done to expand the project, so as to convince the department that it is non-trivial. I have agreed to produce a document listing some of the features of the application, and the general intent of the project.	30m
Feb 10	Produced project proposal v0.1	Extremely early draft of the document Andreas asked for. Essentially just an early list of feature ideas. Eventually this document will turn into a proper project proposal.	30m
Feb 10	Emailed Clive, Sasha, and Nina asking for input on project	Clive and Sasha responded with ideas and comments. Haven't heard from Nina. Clive is working to set up interviews with New Zealand Police and the New Zealand Rural Fire Service. They are representative of end users of the application and will be able to provide valuable insight and information.	15m
Feb 11	Produced project proposal v0.2	Incorporated feedback from yesterday into document, and made some changes of my own.	30m
Feb 12	Began analysing course requirements	Tim Bell posted the Course Information document on Learn today. I highlighted important details. Added due dates to my calendar.	15m

References

[1] J. Torres, A. Lin, J. Rippon, J. Wales, “IndP12: Tait Communications Asset Monitoring within a Vehicle”, ENEL400 Final Year Project Individual Progress Report, University of Canterbury, 2015

[2] A. Curtis-Black, “Assisted Resource Management in the New Zealand Rural Fire Service”, SENG402 Project Proposal, University of Canterbury, 2015

[3] A. Curtis-Black, “Assisted Resource Management in the New Zealand Rural Fire Service”, SENG402 Progress Report 1, University of Canterbury, 2015

[4] A. Curtis-Black, “Assisted Resource Management in the New Zealand Rural Fire Service”, SENG402 Progress Report 2, University of Canterbury, 2015

[5] Apple, Inc. “Apple Developer Documentation”, 2015

[6] A. Curtis-Black, “Assisted Resource Management in the New Zealand Rural Fire Service”, Project Proposal, University of Canterbury, 2015

[7] Trello, Inc. *Trello* [online]. Available: www.trello.com

[8] Apple, Inc. “Compatibility” in *The Swift Blog*, 2014

[9] National Vulnerability Database, *CVE-2014-1266* [online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266

[10] R. Hartson & P. Pyla, “The UX Book: Process and Guidelines for Ensuring a Quality User Experience”, Morgan Kaufmann, 2012